Lecture 4: Introduction to Instruction Set Architecture

Prof Stevens

Fall 2000

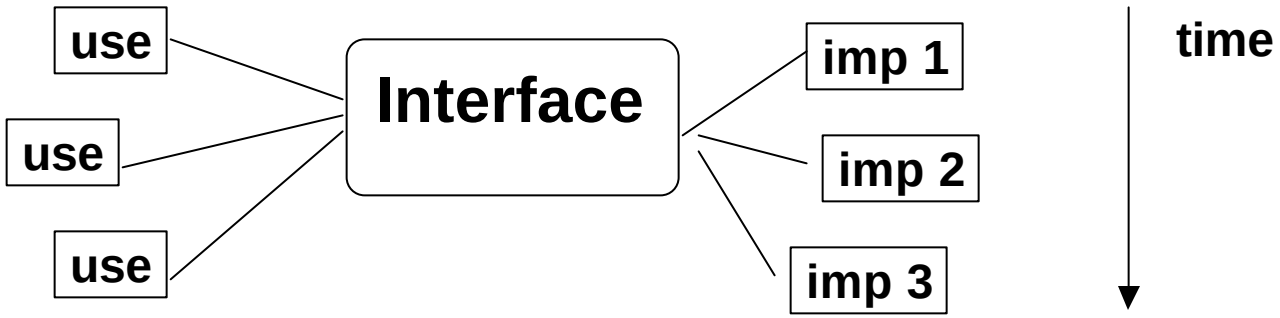
Today: Instruction Set Architecture

- 1950s to 1960s: Computer Architecture Course Computer Arithmetic
- 1970 to mid 1980s: Computer Architecture Course Instruction Set Design, especially ISA appropriate for compilers
- 1990 to 2000s: Computer Architecture Course Design of CPU, memory system, I/O system, Multiprocessors

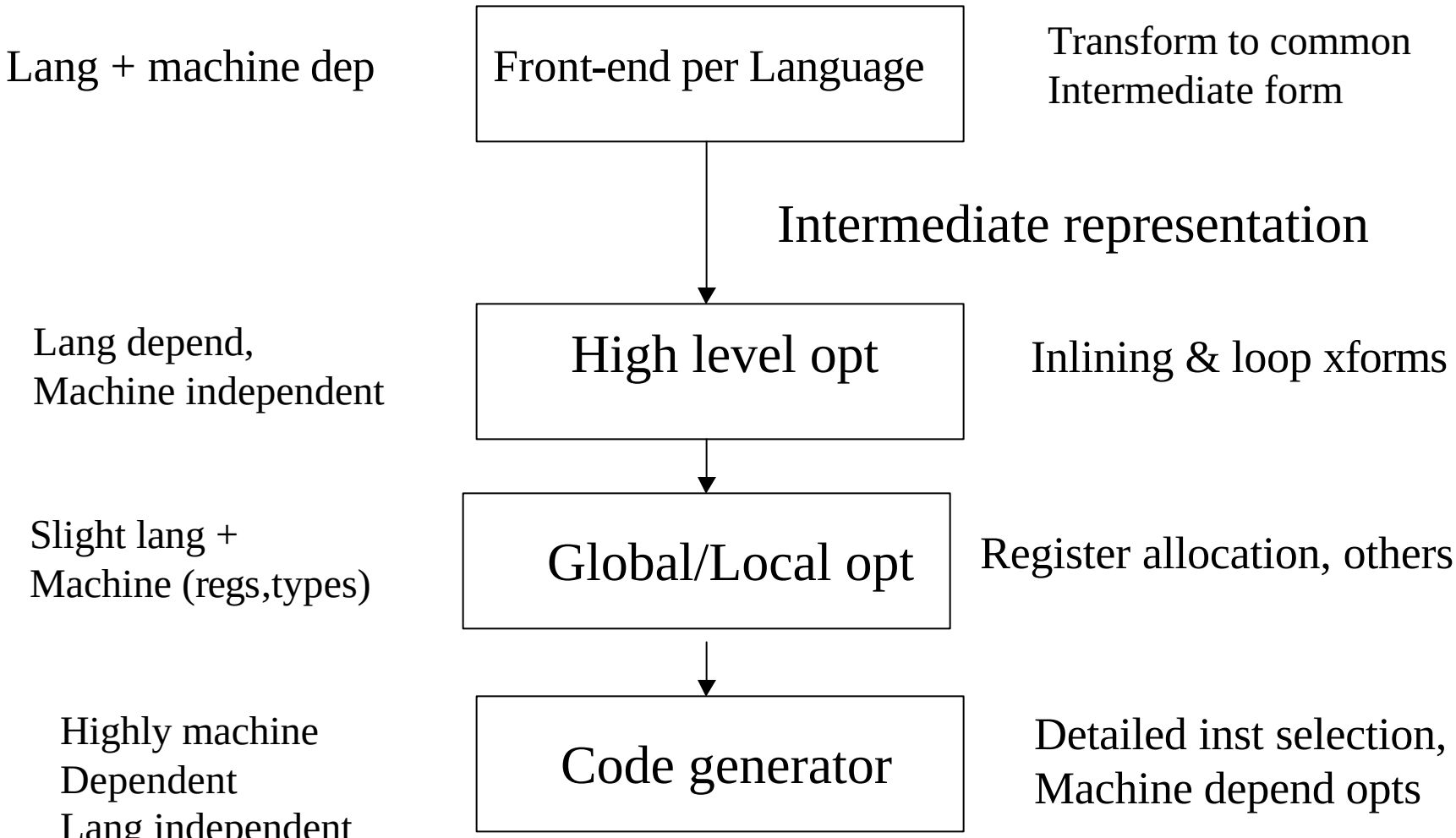
Interface Design

A good interface:

- **Lasts through many implementations (portability, compatibility)**
- **Is used in many different ways (generality)**
- **Provides convenient functionality to higher levels**
- **Permits an efficient implementation at lower levels**



Structure of Modern Compilers



What Is an Instruction Set Anyway?

- Just a set of instructions
  - encoding low level operations
- Each instruction is *directly!!* executed by the CPU's hardware
  - How is it represented?
    - By a binary format since the hardware only *grok's* bits
  - Typical physical blobs are bits, bytes, words, n- words
- Word size is typically 16, 32, 64 bits today
- Options - fixed or variable length formats
  - Fixed - each instruction encoded in same size field - typically 1 word
  - Variable - half- word, whole- word, multiple word instructions are possible

What is an Instruction?

- Usually a simple operation
- Which operation is identified by the *op- code* field
  - But operations require operands - 0, 1, or 2
- To identify where they are, they must be addressed
  - Address is to some piece of storage
- Storage possibilities are main memory, registers, or a stack
    - Each has it's own particular organization
- Two options: explicit or implicit addressing
  - Implicit - the op- code implies the address of the operands
    - ADD on a stack machine - Pops the top 2 elements of the stack, then pushes the result
      - HP calculators work this way
      - Can you spot the pro's and con's of this model?
  - Explicit - the address is specified in some field of the instruction
    - Note the potential for 3 addresses - 2 operands + the destination
  - What are the advantages of addressing registers vs. memory?

What Operations are Needed?

- Arithmetic + Logical
  - ADD, SUB, MULT, DIV, SHIFT - logical and arith, AND, OR, XOR, NOT
- Data Transfer - copy, load, store
- Control - branch, jump, call, return, trap
- System - OS and memory management
  - We'll ignore these for now - but remember they are needed
- Floating Point
  - Same as arithmetic but usually take bigger operands
- Decimal - if you go for it what else do you need?
  - Legacy from COBOL and the commercial application domain
- String - move, compare, search
- Media Processing Instructions, Graphics ?

Design Space of ISA

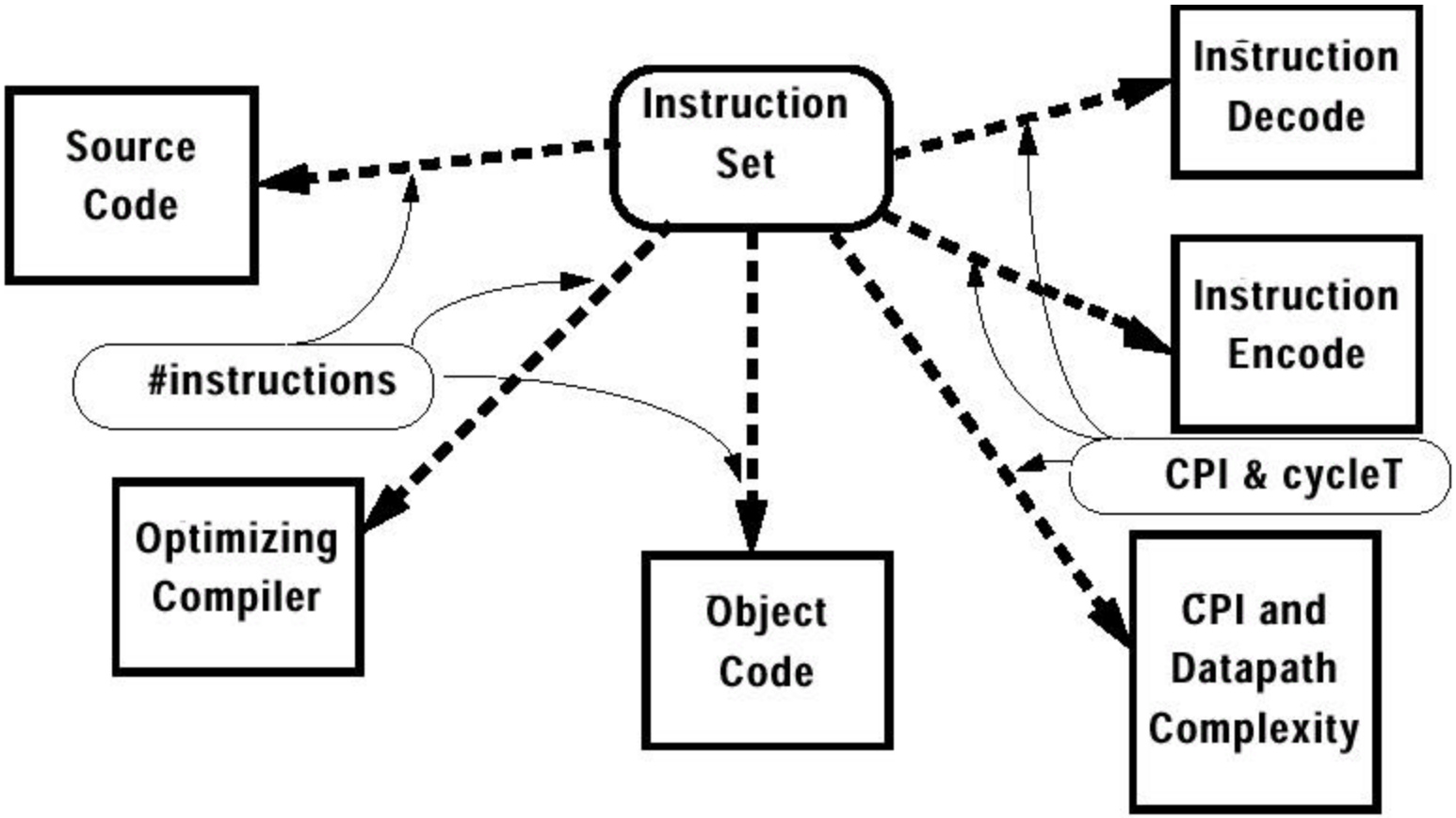
Five Primary Dimensions

• Number of explicit operands	(0, 1, 2, 3)
• Operand Storage	Where besides memory?
• Effective Address	How is memory location specified?
• Type & Size of Operands	byte, int, float, vector, . . .
	How is it specified?
• Operations	add, sub, mul, . . .
	How is it specified?

Other Aspects

• Successor	How is it specified?
• Conditions	How are they determined?
• Encodings	Fixed or variable? Wide?
• Parallelism	

ISA Influence on Performance

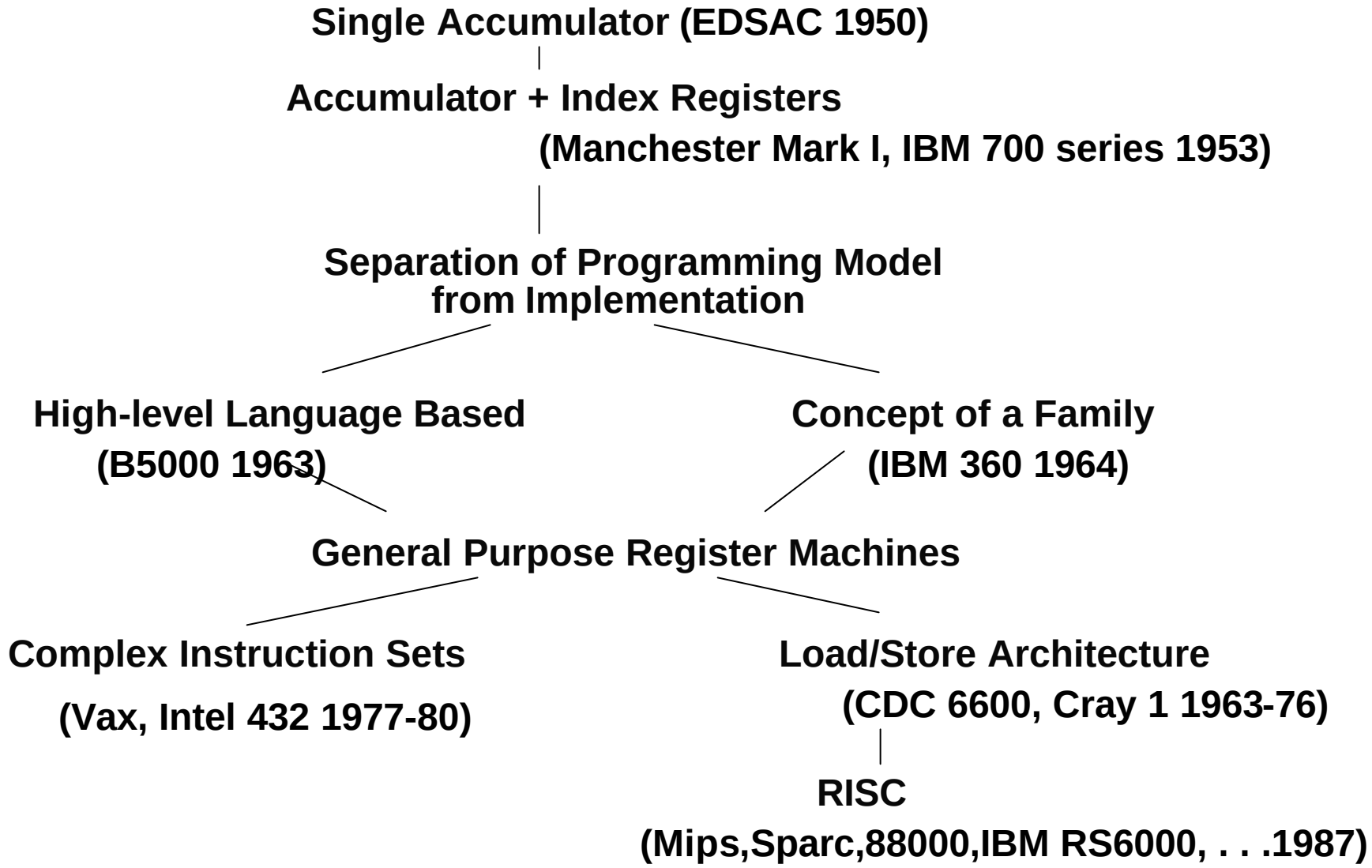


Examples of ISA Types

Storage Type	Examples	Explicit operands per ALU instruc.	Result Destination	Explicit operand access method
Stack	**B5500, B6500 HP2116B HP 3000/70**	**0**	**Stack**	**Push & Pop Stack**
Accumulator	**PDP-8 Motorola 6809 + ancient ones**	**1**	**Accumulator**	**Acc = Acc + mem**
Register Set	**IBM 360 DEC VAX + all modern micro's**	**2 or 3**	**Registers or Memory**	**Rx = Ry + mem (3) Rx = Rx + Ry (2) Rx = Rx + Rz (3)**

register-register, register-memory, and memory-memory (gone) options

Evolution of Instruction Sets



How Do Each Type Compute: C = A + B

- Accumulator
  - **Load A**
  - **Add B**
  - **Store C**
- Stack
  - **Push A**
  - **Push B**
  - **ADD Pop C**
- Register
  - **Load R1, A**
  - **Add R1, B**
  - **Store C, R1**

- Consider - instruction encoding?
- Consider what a search or sort would look like

Pros and Cons of Machine Type

Machine Type	Advantages	Disadvantages
Stack	Simple effective address Short instructions Good code density Simple I-decode	Lack of random access. Efficient code is difficult to generate. Stack is often a bottleneck.
Accumulator	Minimal internal state Fast context switch Short instructions Simple I-decode	Very high memory traffic
Register	Lots of code generation options. Efficient code since compiler has numerous useful options.	Longer instructions. Possibly complex effective address generation. Size and structure of register set has many options.

Look at Fundamental Differences in Datapath and Control- Path

- Assumptions - Datapath
  - Only direct addressing
  - 8 bit opcode for all (not true but difference is minor)
  - 16 registers for GPR machine
  - 16- bit memory address field
  - Ignore byte and half- word accesses for simplicity
  - Pick a 32- bit data- word for grins

- Assumptions - control (ignore conditional- branch)
  - Micro- code like descriptions
  - FSM control is the only control style used today

Instruction Format

Accumulator - 24 bit instruction

8-bit OpCode	16-bit Memory Address

Stack - also 24 bit but most instructions will be 8-bit (pack-em)

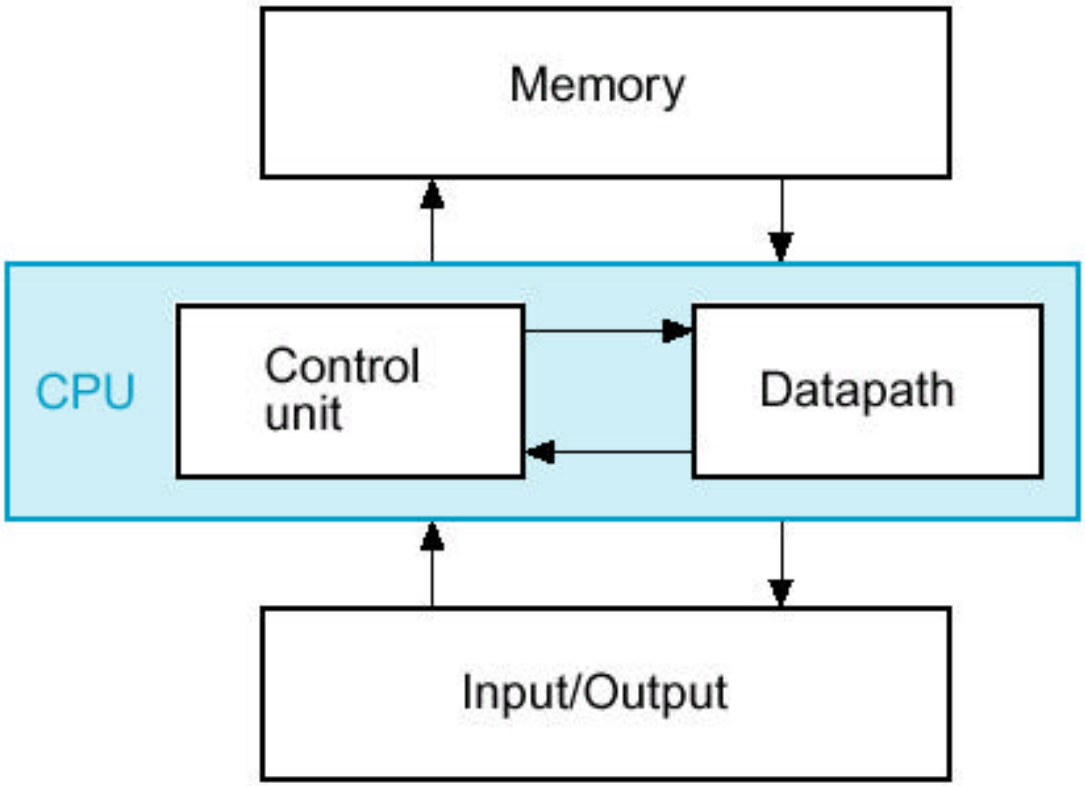
8-bit OpCode	16-bit Memory Address

Register - 2 explicit operands (3 explicit is obvious) - 28 bits

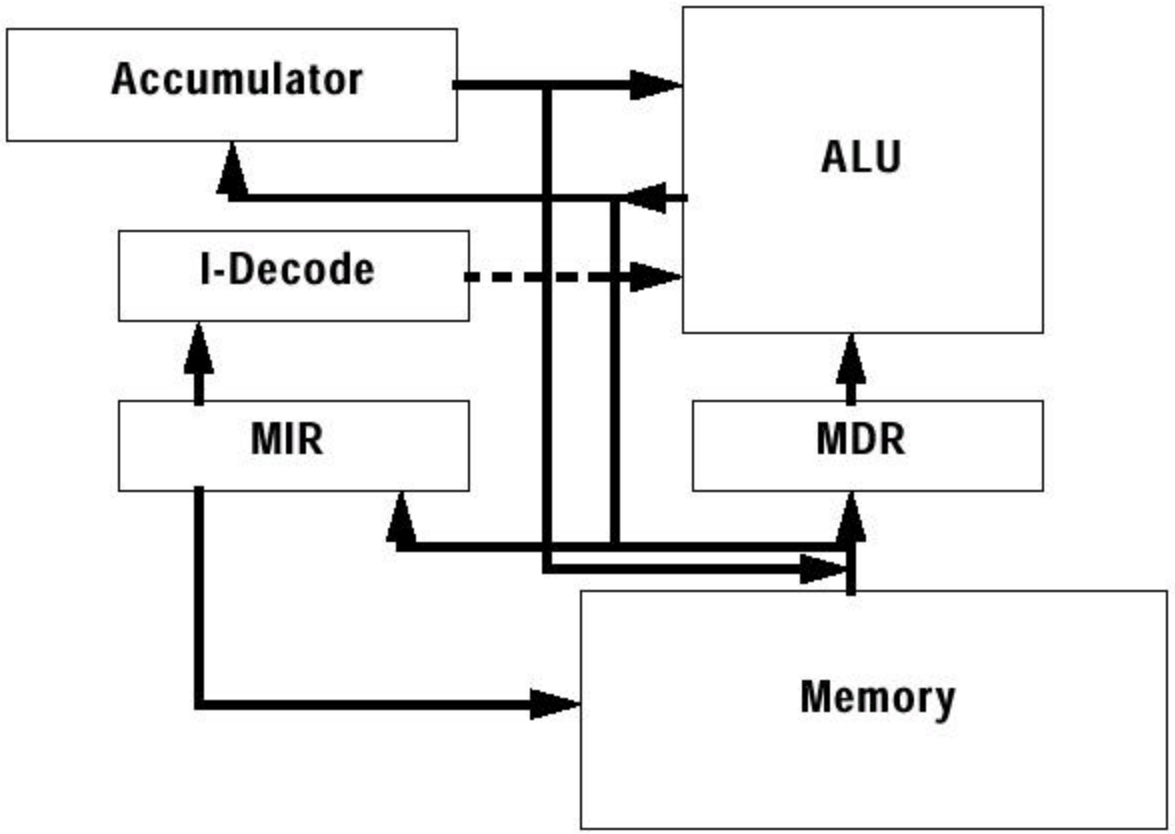
8-bit OpCode	4-bit Reg-addr	16-bit Memory Address

- Could pack but instruction word allignment would be a problem.

Basic System Structure



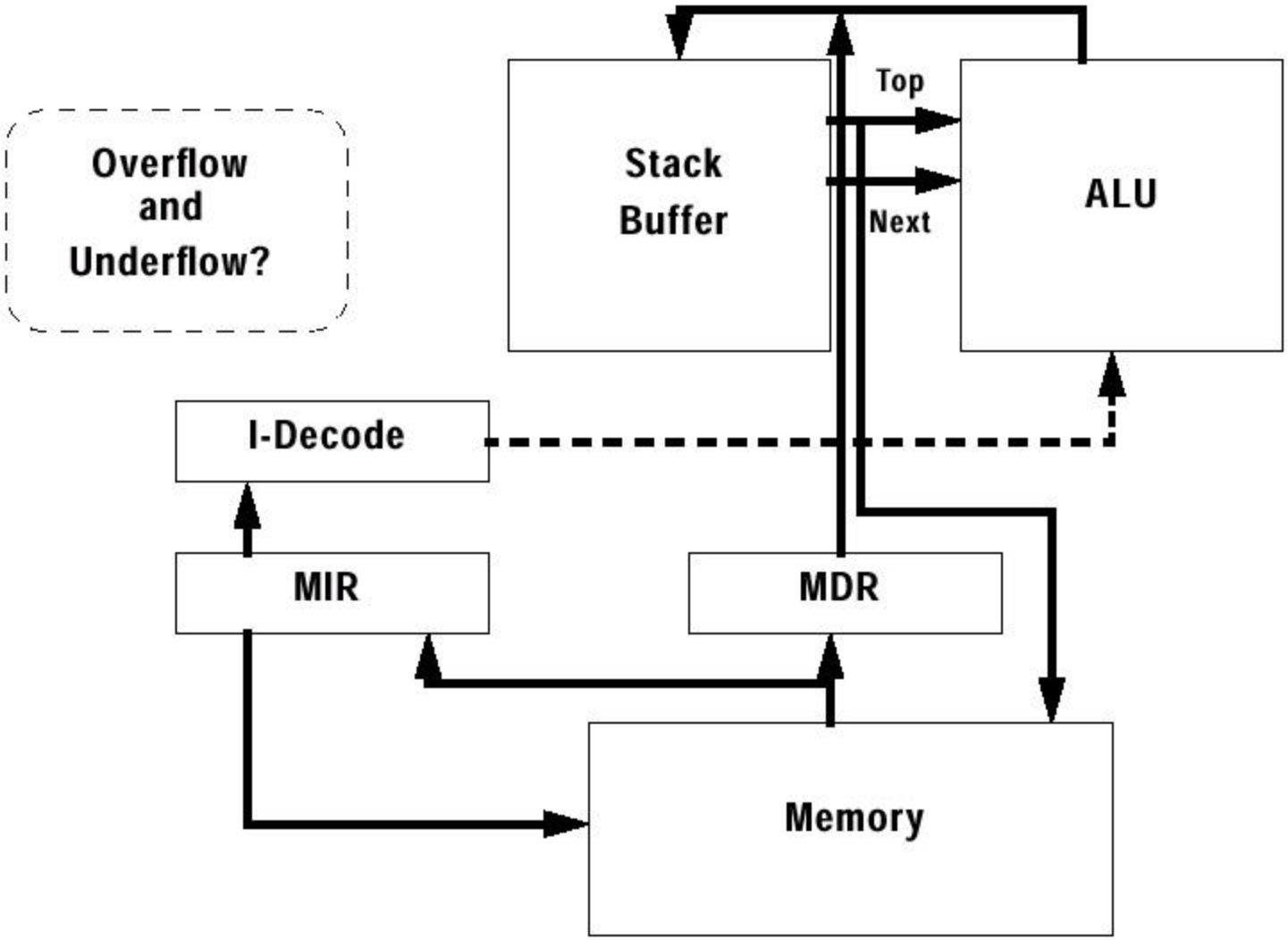
Simple Accumulator Machine



Accumulator Control

- Loads
  - Read Memory, Enable Memory to Accumulator
  - Load Accumulator
- Stores
  - Enable Accumulator to MBUS
  - Write Memory
- ALU Op's
  - Read Memory
  - Enable ALU to Accumulator
  - Load Accumulator
- Branch - just like an IFetch but with PC as address source
  - Read Memory
  - Enable Memory to MIR
  - Load MIR

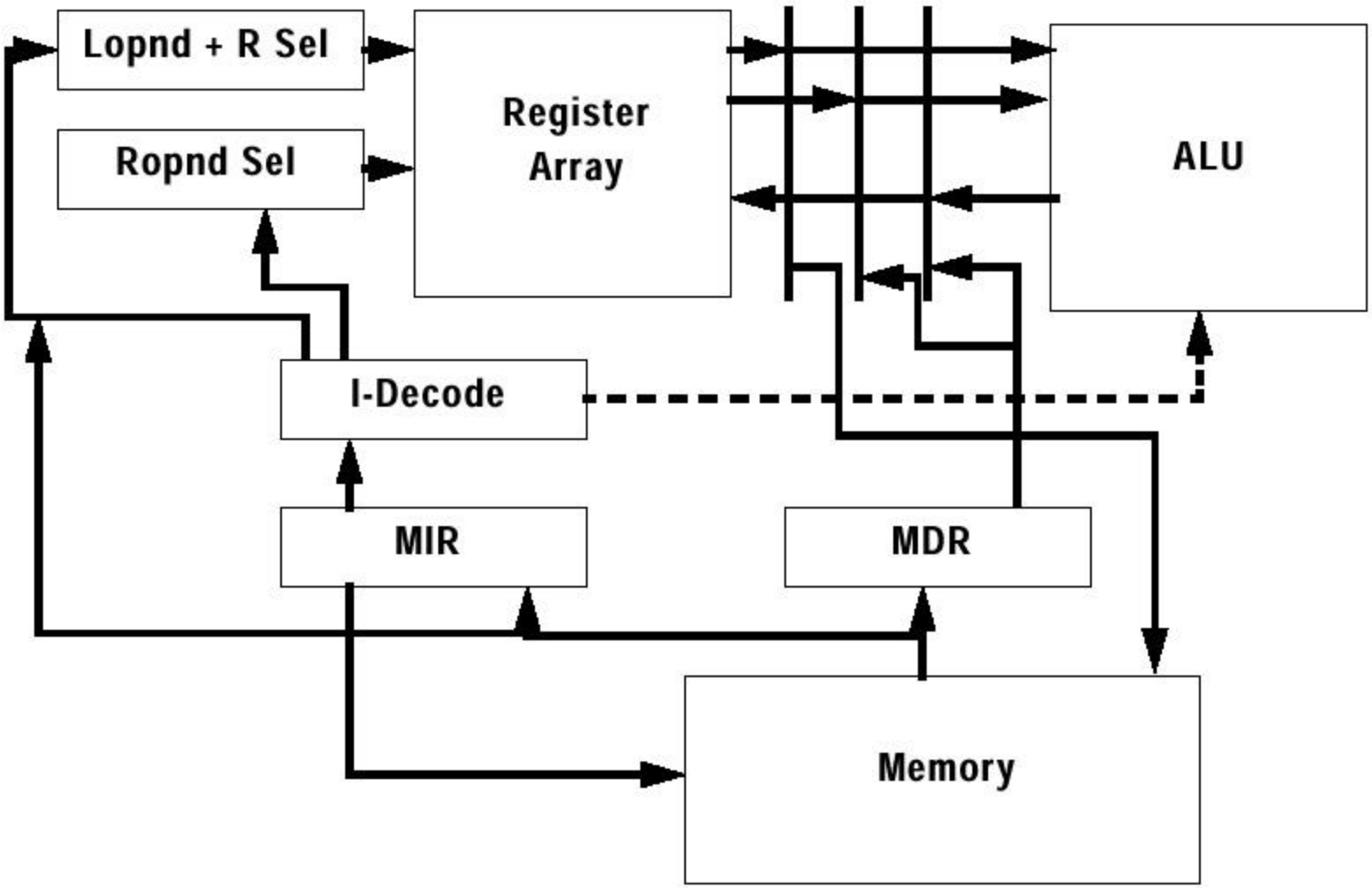
Basic Stack Machine



Stack Control (over- simplified)

- Loads
  - Read Memory
  - Push
- Stores
  - Enable Top to MBUS, Write Memory
  - Pop
- ALU Op’s
  - Load Top or Next
  - Pop or not
- Branch - just like an IFetch but with PC as address source
  - Read Memory
  - Enable Memory to MIR
  - Load MIR

General Purpose Register Machine



GPR Machine Control

- Loads - just like accumulator but select Reg.
- Stores - just like accumulator but select Reg.
- Branch - same as all the rest
- ALU OPs - whoa!
  - Select Left operand and result register
  - Decide whether you want memory or a register for the right operand
  - Note minimum of 3 busses between the Register array and the ALU

GPR Machines Won!

- How many Reg's vs. which structure is the debate of the day
- Lot's of options have gone to press Hence Register (mem, operand) classification

# Memory Ops per typical ALU instruction	Max ALU operands allowed	Examples
0	2 3	IBM RT-PC SPARC, MIPS, HP-PA, PowerPC, ALPHA
1	2 3	PDP-10, M6800, IBM 360, Intel 90x86 IBM 360RS
2	2 3	PDP-11, National 32x32, IBM 360SS, VAX NEC S1
3	3	VAX - blech!

Register- Register (0, 3) Pro's + Con's

- (m, n) m memory operands, n total operands in ALU instruction
- ALU is Register to Register -- i. e. pure RISC
- Advantages:
  - Simple fixed length instruction encoding
  - Decode is simple since instruction types are small
  - Simple code generation model
  - Instruction CPI tends to be very uniform
- Disadvantages:
  - Instruction count tends to be higher
  - Some instructions are short - wasting instruction word bits

Register- Memory (1, 2) P + C's

- Evolved RISC and also old CISC - go figure?
  - New RISC machines capable of doing speculative and/ or deferred loads
- Register - Memory ALU Architecture
- Advantages:
  - Data access to ALU immediate without loading first
  - Instruction format is relatively simple
  - Density is improved over Register (0, 3) model
- Disadvantages:
  - Operands are not equivalent - source operand may be destroyed
  - Need for memory address field may limit # of registers
  - CPI will vary

Memory- Memory (3, 3) P + C's

- True Memory - Memory ALU Architecture
- True and most complex CISC model
  - currently extinct and likely to remain so
  - more complex memory actions are likely to appear but not directly linked to the ALU
- Advantages:
  - Most compact
  - Doesn't waste registers for temporary values
- Disadvantages:
  - Large variation in instruction size - may need a shoe- horn
  - Large variation in CPI - i. e. work per instruction
  - Exacerbates the infamous memory bottleneck

Memory Addressing

- All architectures clearly need some way to address memory
- **A number of questions naturally arise**
- What is accessed - byte, word, multiple words
  - – • For no good reason many of today's machine are byte addressable
  - – • But the main memory is organized in 32 - 64 byte lines to match cache model
- Hence there is a natural alignment problem
  - – Accessing a word or double- word which crosses 2 lines requires 2 references
  - – automagic alignment is possible but hides the number of references
  - – Also therefore hides an important case of CPI bloat hence a bad idea

When We Address Bytes – Who Gets to Be First?

- Same issue with bit numbers - less of an ISA issue

- Big Endian - byte 0 is the MSB
- Little Endian - byte 0 is the LSB

- Wouldn’t be a big deal if we didn’t have byte serial communication and I/ O devices

Other Alignment Issues

- Common convention is to expect aligned data objects
  - compiler is responsible for keeping this straight
  - hardware just checks - e. g. generates a trap if alignment restrictions are violated
  - implies that op- code is type specific LDB - load byte, LDW - load word, etc.
- Hence
  - byte address is anything - never misaligned
  - half word - even addresses - low order address bit = 0 else trap
  - word - low order 2 address bits = 0 else trap
  - double word - low order 3 address bits = 0 else trap

Addressing Modes

- An important aspect of ISA design
  - has major impact on both the HW complexity and the IC
  - HW complexity affects the CPI and the cycle time
- Basically a set of mappings
  - from address specified to address used
  - address used = *effective address*
  - Effective address may go to memory or to a register array which is typically dependent on it's location in the instruction field
  - in some modes multiple fields are combined to form a memory address
  - register addresses are usually more simple - e. g. they need to be fast effective address generation is an important focus since it is the common case - e. g. every instruction needs it it must also be fast
- Note tower of Babel effect
  - address mode names are vendor specific

Addressing Modes I

Mode	Example Instruction	Meaning	Use
Register	Add R4, R3	Regs[R4] <- Regs[R4] + Regs[R3]	All RISC ALU operations
Immediate	Add R4, #3	Regs[R4] <- Regs[R4] + 3	for small constants - problems?
Displacement	Add R4, 100(R1)	Regs[R4] <- Regs[R4] + Mem[100 + Regs[R1]]	accessing local variables
Register deferred or Indirect	Add R4, (R1)	Regs[R4] <- Regs[R4} + Mem[Regs[R1]]	pointers
Indexed	Add R3, (R1 + R2)	Regs[R3] <- Regs [R3] + Mem[Regs[R1] + Regs[R2]]	array access - R1 is the base, R2 is the index
Direct or absolute	Add R1, (1001)	Regs[R1] <- Regs[R1] + Mem[1001]	

Addressing Modes II

Mode	Example Instruction	Meaning	Use
Memory Indirect or Memory Deferred	Add R1, @R3	Regs[R1] <- Regs[R1] + Mem[Mem[Regs[3]]]	If R3 holds a pointer address, then result is the full dereferenced pointer
Autoincrement in this case post increment note symmetry with autodec	Add R1, (R2) +	Regs[R1] <- Regs[R1] + Mem[Regs[R2]]; Regs[R2] <- Regs[R2] + d	Array walks - if element of size d is accessed then pointer increments auto
Autodecrement in this case predecrement	Add R1, - (R2)	Regs[R2] <- Regs[R2] - d; Regs[R1] <- Regs[R1] + Mem[Regs[R2]];	array walks, with autoinc useful for stack implementation
Scaled	Add R1, 100 (R2) [R3]	Regs[R1] <- Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]	array access - may be applied to indexed addressing in some machines

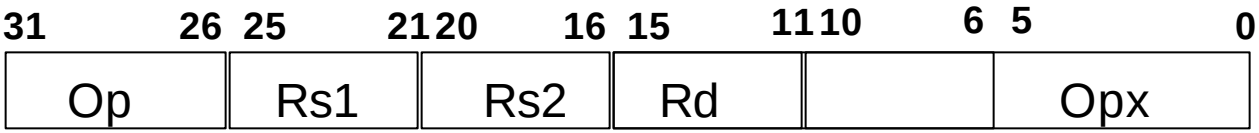
A "Typical" RISC

- 32-bit fixed format instruction (3 formats)
- 32 32-bit GPR (R0 contains zero, DP take pair)
- 3-address, reg-reg arithmetic instruction
- Single address mode for load/store:
  base + displacement
  - no indirection
- Simple branch conditions
- Delayed branch

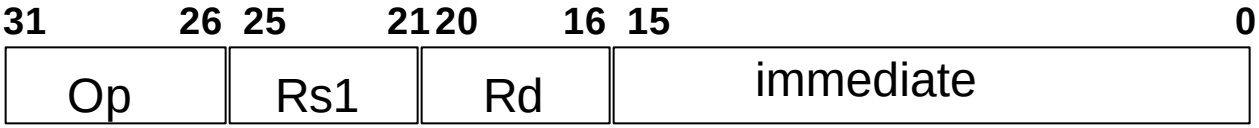
see: SPARC, MIPS, MC88100, AMD2900, i960, i860
PARisc, DEC Alpha, Clipper,
CDC 6600, CDC 7600, Cray-1, Cray-2, Cray-3

Example: MIPS

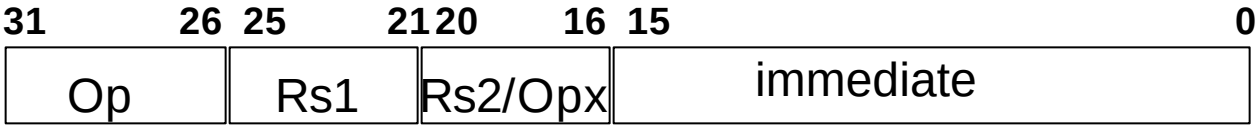
Register-Register



Register-Immediate



Branch



Jump / Call

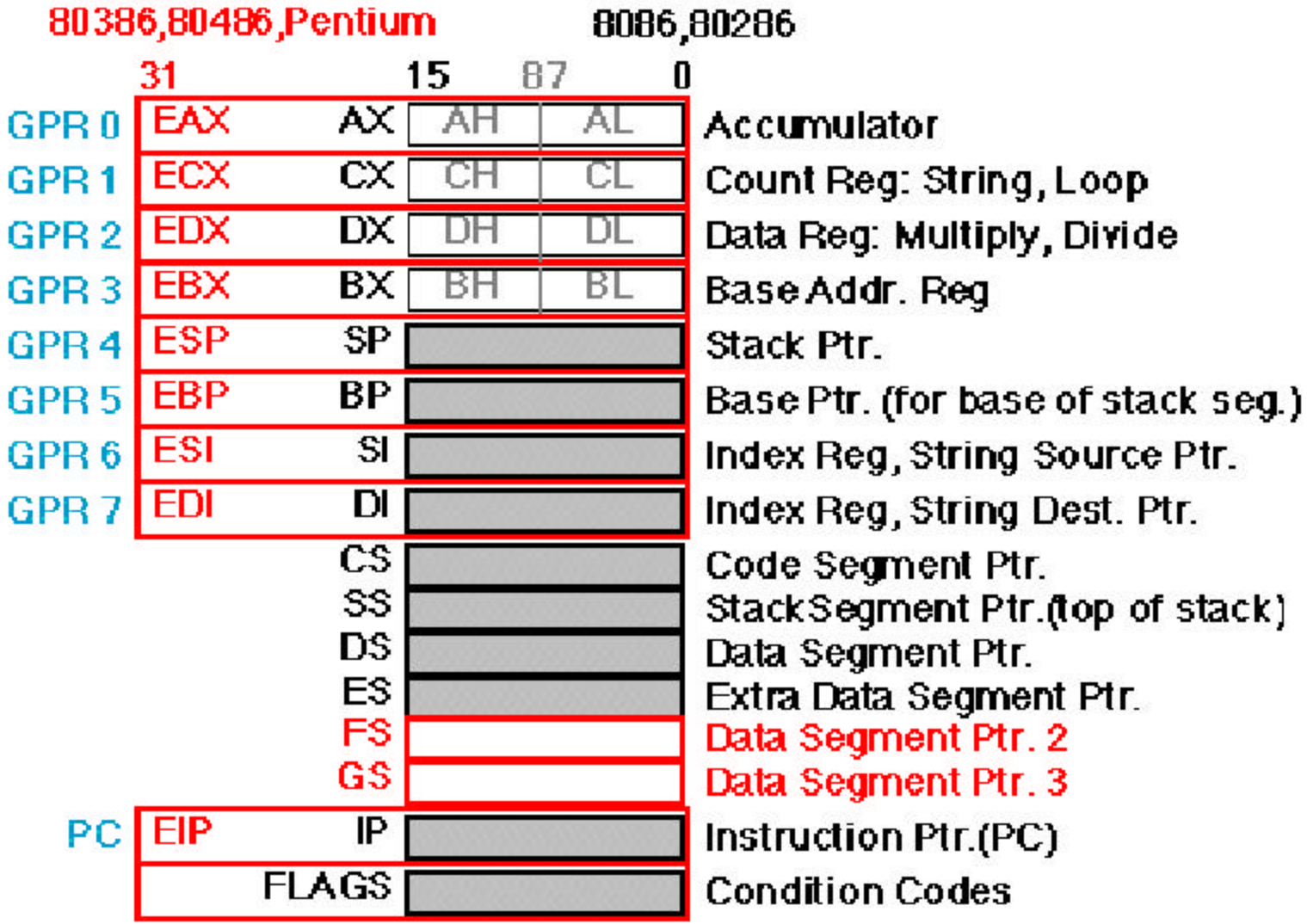
Most Popular ISA of All Time: The Intel 80x86

- 1971: Intel invents microprocessor 4004/8008,
  - 8080 in 1975
- 1975: Gordon Moore realized one more chance for new ISA before ISA locked in for decades
  - Hired CS people in Oregon
  - Weren't ready in 1977
    - CS people did 432 in 1980
  - Started crash effort for 16-bit microcomputer
  - 1978: 8086 dedicated registers, segmented address, 16 bit
  - 8088; 8-bit external bus version of 8086; added as after thought

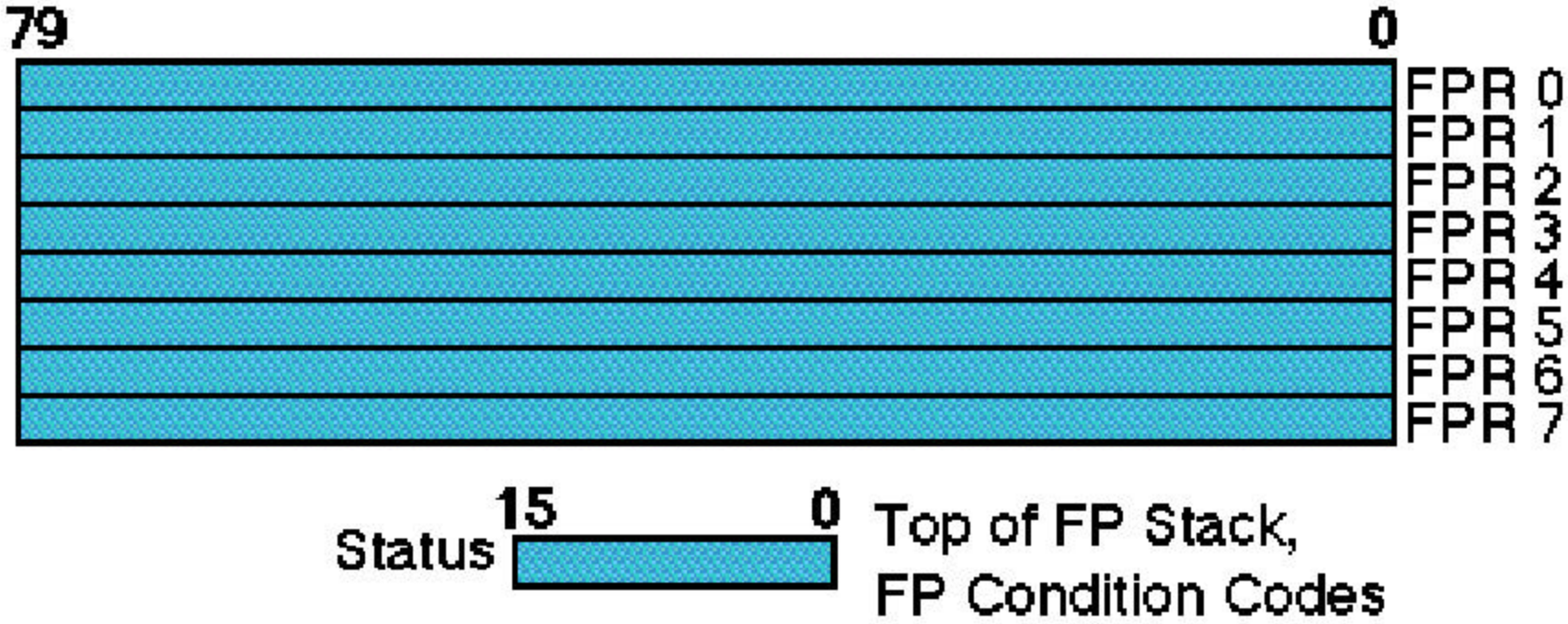
Most Popular ISA of All Time: The Intel 80x86

- 1980: IBM selects 8088 as basis for IBM PC
- 1980: 8087 floating point coprocessor: adds 60 instructions using hybrid stack/register scheme
- 1982: 80286 24-bit address, protection, memory mapping
- 1985: 80386 32-bit address, 32-bit GP registers, paging
- 1989: 80486 & Pentium in 1992: faster + MP few instructions

Intel 80x86 Integer Registers

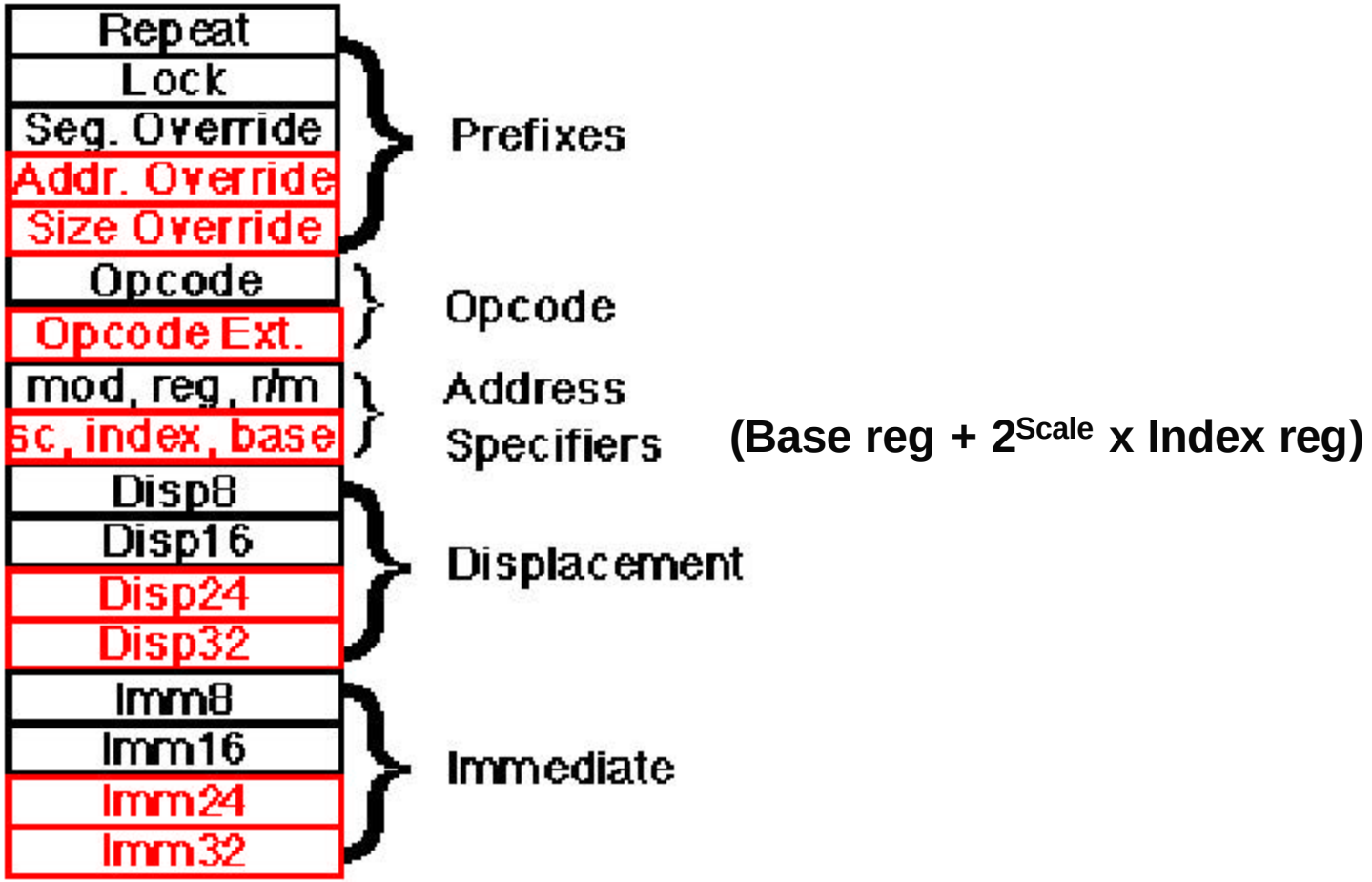


Intel 80x86 Floating Point Registers



80x86 Instruction Format

- 8086 in black; 80386 extensions in color



80x86 Instruction Encoding: Mod, Reg, R/M Field

r	w=0	w=1		r/m	mod=0		mod=1		mod=2		mod=3
		16b	32b		16b	32b	16b	32b	16b	32b	
0	AL	AX	EAX	0	addr=BX+SI	=EAX	*same*	*same*	*same*	*same*	*same*
1	CL	CX	ECX	1	addr=BX+DI	=ECX	*addr*	*addr*	*addr*	*addr*	*as*
2	DL	DX	EDX	2	addr=BP+SI	=EDX	*mod=0*	*mod=0*	*mod=0*	*mod=0*	*reg*
3	BL	BX	EBX	3	addr=BP+SI	=EBX	*+d8*	*+d8*	*+d16*	*+d32*	*field*
4	AH	SP	ESP	4	addr=SI	=(sib)	SI+d8	(sib)+d8	SI+d8	(sib)+d32	“
5	CH	BP	EBP	5	addr=DI	=d32	DI+d8	EBP+d8	DI+d16	EBP+d32	“
6	DH	SI	ESI	6	addr=d16	=ESI	BP+d8	ESI+d8	BP+d16	ESI+d32	“
7	BH	DI	EDI	7	addr=BX	=EDI	BX+d8	EDI+d8	BX+d16	EDI+d32	“

r/m field depends on mod and machine mode

w from opcode

First address specifier: Reg=3 bits, R/M=3 bits, Mod=2 bits

80x86 Instruction Encoding
Sc/Index/Base field

	Index	*Base*
0	EAX	EAX
1	ECX	ECX
2	EDX	EDX
3	EBX	EBX
4	no index	ESP
5	EBP	if mod=0, d32 if mod°0, EBP
6	ESI	ESI
7	EDI	EDI

Base + Scaled Index Mode

Used when:

- **mod = 0,1,2**
- **in 32-bit mode**
- **AND r/m = 4!**

2-bit Scale Field

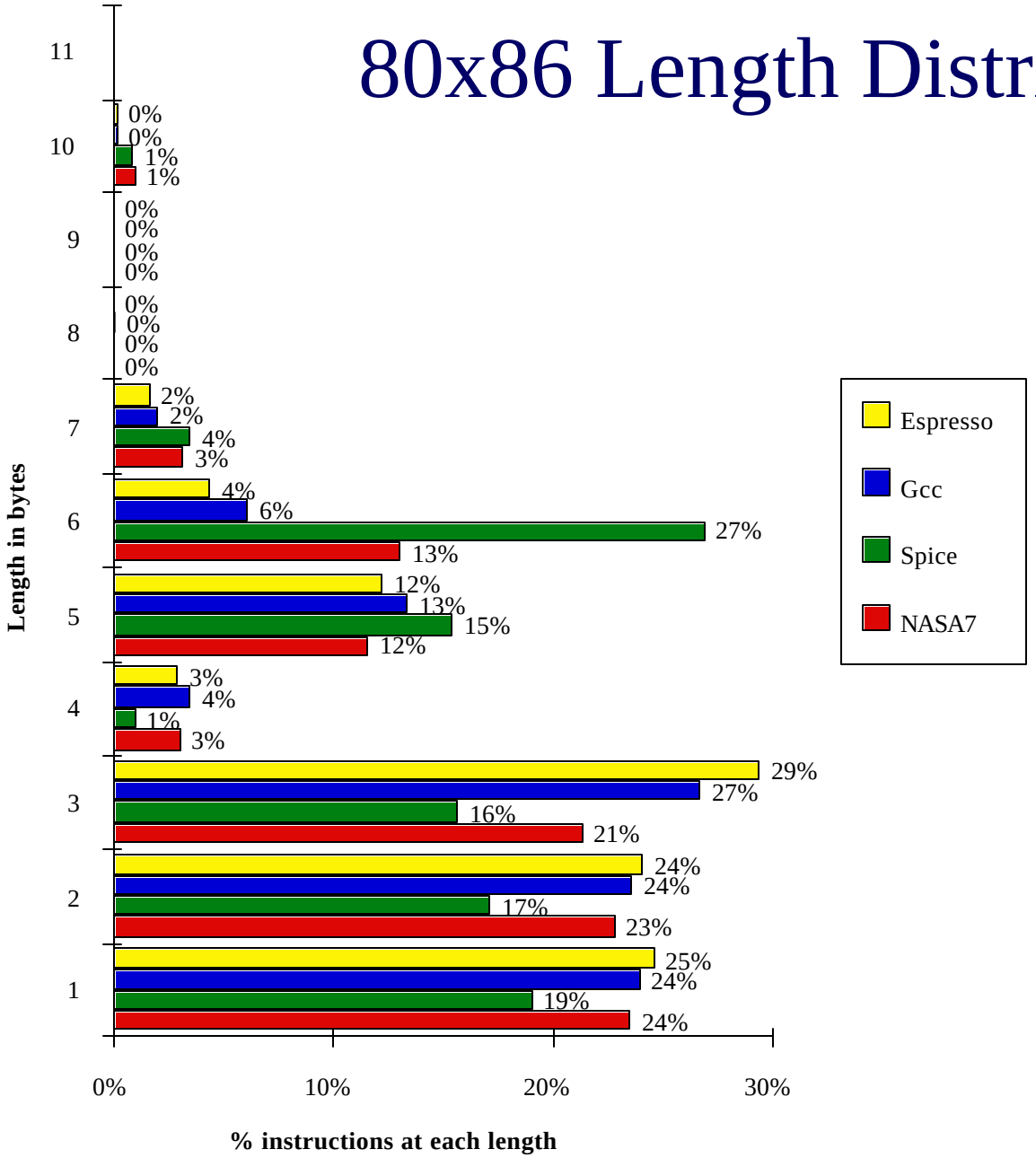
3-bit Index Field

3-bit Base Field

80x86 Addressing Mode Usage for 32-bit Mode

Addressing Mode	*Gcc*	*Espr.*	*NASA7*	*Spice*	*Avg.*
Register indirect	10%	10%	6%	2%	7%
Base + 8-bit disp	46%	43%	32%	4%	31%
Base + 32-bit disp	2%	0%	24%	10%	9%
Indexed	1%	0%	1%	0%	1%
Based indexed + 8b disp	0%	0%	4%	0%	1%
Based indexed + 32b disp	0%	0%	0%	0%	0%
Base + Scaled Indexed	12%	31%	9%	0%	13%
Base + Scaled Index + 8b disp	2%	1%	2%	0%	1%
Base + Scaled Index + 32b disp	6%	2%	2%	33%	11%
32-bit Direct	19%	12%	20%	51%	26%

80x86 Length Distribution



Instruction Counts: 80x86 v. DLX

SPEC pgm	*x86*	*DLX*	*DLX÷86*
gcc	3,771,327,742	3,892,063,460	1.03
espresso	2,216,423,413	2,801,294,286	1.26
spice	15,257,026,309	16,965,928,788	1.11
nasa7	15,603,040,963	6,118,740,321	0.39

Intel Summary

- Archeology: history of instruction design in a single product
  - Address size: 16 bit vs. 32-bit
  - Protection: Segmentation vs. paged
  - Temp. storage: accumulator vs. stack vs. registers
- “Golden Handcuffs” of binary compatibility affect design 20 years later, as Moore predicted
- Not too difficult to make faster, as Intel has shown
- Does the IA-64 announcement of common future instruction means end of 80x86???
- “Beauty is in the eye of the beholder”
  - At 120M/year sold, it is a beautiful business